INTERPOSING A VIRTUAL MACHINE MONITOR AND DEVIRTUALIZING COMPUTER HARDWARE

BACKGROUND

[0001] A virtual machine monitor ("VMM") creates an environment that allows multiple operating systems to run simultaneously on the same computer hardware. In such an environment, applications written for different operating systems (e.g., Windows, Linux) can be run simultaneously on the same hardware.

[0002] When an operating system ("OS") is run on a VMM, unprivileged instructions of the operating system execute on the hardware at full hardware speed. However, most or all instructions that access a privileged hardware state trap to the VMM. The VMM simulates the execution of those instructions as needed to maintain the illusion that the operating system has sole control over the hardware on which it runs.

[0003] I/O handling involves two levels of device drivers for each device: one maintained by the VMM, and the other maintained by the operating system. When an application requests the operating system to perform an I/O function, the operating system invokes a device driver. That device driver then invokes the corresponding device driver maintained by the VMM to perform the I/O function. Similarly, when an I/O interrupt comes in, a VMM device driver handles the incoming interrupt and may deliver it to the corresponding device driver maintained by the operating system.

[0004] The VMM typically handles memory by managing memory translation in order to translate between the OS's use of physical memory, and the real "machine" memory present in hardware.

[0005] The VMM adds to the overhead of the computer. Adding the VMM's management of memory to the OS's own memory management slows memory access. The two layers of device drivers add to the overhead by increasing the amount of software that processes I/O requests and interrupts. Overhead is also added by constantly trapping and simulating privileged instructions, and by forcing I/O requests to go through two levels of device drivers. This overhead can slow interrupt handling, increase the fraction of CPU bandwidth lost to software overhead, increase response time, and decrease perceived performance.

[0006] The VMM is loaded during bootup of the computer and receives control of the hardware at boot time. The VMM maintains hardware control until the computer is shut down.

[0007] Since the VMM has hardware control from bootup to shutdown, overhead is incurred even when the VMM is not needed (for example, when only a single OS instance is running on the hardware). Thus the VMM can add unnecessary overhead to the computer.

[0008] It would be desirable to reduce the unnecessary overhead.

SUMMARY

[0009] According to one aspect of the present invention, a virtual machine monitor is interposed between computer hardware and an operating system at runtime. According to another aspect of the invention, at least some of the hardware is devirtualized at runtime. Other aspects and advantages of the present invention will become apparent from the following detailed description, taken in conjunction with the accompanying drawings, illustrating by way of example the principles of the present invention.

BRIEF DESCRIPTION OF THE DRAWINGS

[0010] FIG. 1 is an illustration of hardware and software layers of a computer in accordance with an embodiment of the present invention.

[0011] FIGS. 2a and 2b are illustrations of methods of using a virtual machine monitor in accordance with different embodiments of the present invention.

[0012] FIG. 3 is an illustration of a method of interposing a virtual machine monitor on a CPU in accordance with an embodiment of the present invention.

[0013] FIG. 4 is an illustration of a method of loading a virtual machine monitor into memory at runtime.

[0014] FIG. 5 is an illustration of a method of interposing a virtual machine monitor on an I/O device in accordance with an embodiment of the present invention.

[0015] FIG. 6 is an illustration of a method of returning control of an I/O device to an operating system in accordance with an embodiment of the present invention.

[0016] FIG. 7 is an illustration of a method of devirtualizing a CPU in accordance with an embodiment of the present invention.

[0017] FIG. 8 is an illustration of a method of removing or disabling a VMM in accordance with an embodiment of the present invention.

DETAILED DESCRIPTION

[0018] As shown in the drawings for purposes of illustration, the present invention is embodied in a computer that can run a virtual machine monitor. The computer is not limited to any particular type. The computer can be, for example, a file server, web server, workstation, mainframe, personal computer, personal digital assistant (PDA), print server, or network appliance. The computer can be contained in a single box, or distributed among several boxes.

[0019] Reference is made to FIG. 1a, which illustrates hardware and software layers 110 and 111 of a computer 100. The hardware layer 110 includes a central processing unit (CPU), memory and I/O devices. Exemplary I/O devices include, without limitation, network adapters, SCSI controllers, video cards, host bus adapters, and serial port adapters. The memory refers to the memory that is internal to the computer (e.g., internal memory cache, main system memory) as opposed to external storage devices such as disk drives. The software layer 111 includes an operating system or OS instances 112, applications 114, and a virtual machine monitor 116. During execution, the software for the software layer 111 can be stored in "articles" such as the memory; and during distribution, the software can be stored in articles such as external devices, removable storage media (e.g., optical discs), etc.

[0020] Additional reference is made to FIG. 2a, which illustrates a method of using the VMM 116. The operating system 112 is loaded during bootup of the computer (210), and takes control of the hardware layer 110 at boot time. As a result, the operating system 112 has direct hardware control over the CPU, the memory, and the I/O devices. The applications 114 may be run on the operating system 112.

[0021] The virtual machine monitor 116 is interposed between the hardware layer 110 and the operating system 112 at runtime (212). Runtime is the period of normal execution of the operating system after boot and before shutdown. Interposing the VMM 116 gives the VMM 116 direct control of at least one of the CPU, the memory and the I/O devices. While the VMM 116 has direct control of the CPU, the CPU is said to be "virtualized." Similarly, those portions of physical memory directly controlled by the VMM 116 are said to be virtualized, and those I/O devices directly controlled by the VMM 116 are said to be virtualized. After being interposed, the VMM 116 can then be used for its intended purpose, such as running additional operating system instances. These operating system instances can share the virtualized hardware.

[0022] The VMM 116 can be interposed only when needed, which can occur long after the computer has booted. As a result, overhead is reduced, since the VMM 116 does not add to the overhead between computer bootup and actual use.

[0023] Once interposed, however, the virtual machine monitor adds overhead to the computer. This overhead can become unnecessary once the VMM 116 is not used for its intended purpose.

[0024] This unnecessary overhead can be reduced by devirtualizing the hardware layer 110 (214). Devirtualizing the hardware 110 gives the operating system 112 direct hardware control over at least one of the CPU, the physical memory, and the I/O. The hardware layer 110 may be partially devirtualized, whereby the memory alone is devirtualized, or the I/O alone is devirtualized, or the memory and I/O are devirtualized. If the hardware 110 is fully devirtualized (that is, the operating system 112 is given direct control

over the CPU, the physical memory, and the I/O), the VMM can be unloaded from the computer 100.

[0025] Reference is made to FIG. 2b, which illustrates an alternative embodiment of a method of using the VMM 116. The VMM 116 is loaded at bootup of the computer (250), and an operating system is booted on the VMM (252). The hardware layer 110 is then devirtualized as soon as the VMM 116 is not needed (254). Devirtualization may be full or partial. For example, the CPU could remain virtualized, but memory and I/O would be devirtualized. The VMM 116 is not unloaded from memory even if the hardware is fully devirtualized. When needed, the VMM 116 is interposed between the operating system and the hardware (256). When the VMM 116 is no longer needed, the hardware layer 110 is devirtualized, either fully or partially. Relative strengths of these two embodiments will become apparent below.

[0026] The steps in FIGS. 2a and 2b can be performed or initiated by a system administrator. In the alternative, these steps may be performed or initiated in software by an application, a script, etc.

[0027] Reference is now made to FIG. 3, which illustrates an exemplary method of interposing the VMM 116 at runtime on a CPU. First, the VMM is loaded into memory (310) if it has not been loaded already. This step (310) is not performed if the VMM has already been loaded in advance (e.g. at boot time) without virtualizing the hardware, or if it has been loaded into memory from a prior period in which the VMM virtualized the CPU (e.g., the method shown in FIG. 1b).

[0028] If necessary, an OS kernel can be used to invoke an initialization routine in the VMM (312). This routine may go through resource discovery to discover the hardware that is installed in the computer. The initialization routine may also initialize internal VMM data structures and

device drivers, and it may also carry out at least one of the following steps (314-324).

[0029] Next interrupts are disabled (314). If the VMM has sufficient privilege, it can disable the interrupts. For example, the computer 100 might allow the VMM to disable the interrupt, or the VMM might have been previously booted/devirtualized and retained sufficient privilege to disable the interrupts. If the VMM cannot disable the interrupts, an OS kernel module can be invoked to perform this step.

[0030] Interrupts are redirected to handlers in VMM so the VMM gets control on interrupts and traps (316). This step will likely involve modifying the interrupt vector table to invoke VMM handlers rather than OS handlers. Interrupt handlers for the CPU typically include NMI, machine check, timer, interprocessor, and cycle counter.

[0031] After the interrupts have been redirected, interrupts are re-enabled (318). After that, the direct addressability of physical memory is disabled (320). Completing this latter step (320) will allow the VMM to manage (e.g. trap or map) access to physical addresses.

[0032] Next, privileged instructions are caused to trap to the VMM (322). On some architectures, causing the privileged instructions to trap to the VMM may involve reducing the current processor mode of the CPU (i.e. its privilege level). On other architectures, it may not be necessary to reduce the CPU's privilege level much or at all. For example, on the HP Alpha architecture, the OS usually runs in a fairly unprivileged mode relative to the PALcode (a privileged library between the hardware and operating system). If the VMM is implemented using the privilege afforded the PALcode, then it is not necessary to further reduce the privilege of the OS. For architectures such as Intel x86 and IA-64, causing the OS's privileged instructions to trap to

the VMM may involve modifying the OS's executable image in memory. For example, the VMM replaces some instructions that can reveal privileged state without trapping to the VMM. The replacement instructions may instead invoke a routine in the VMM. For some architectures, the VMM may also modify aspects of the OS code to optimize the OS performance.

[0033] Control is returned to the OS at this reduced privileged level (324). If the CPU alone is virtualized, the operating system will still have direct control over the physical memory and the I/O devices.

[0034] If the VMM is loaded into memory at boot time, firmware or a boot loader can set aside sufficient physical memory for the VMM before the OS boots, and then load the VMM into that range of memory. The firmware or boot loader should shield the booting OS from discovering the range of memory set aside for the VMM, for example, by modifying the table passed to the OS describing the physical memory available for its use.

[0035] Reference is made to FIG 4, which illustrates a method of using a kernel module to load the VMM into memory at runtime. First, the kernel module allocates sufficient memory using the kernel module's internal memory allocator (410). Next, the kernel module "pins" those pages of memory to prevent the OS from taking them back prematurely (412). The module may perform the pinning of memory using the OS's internal routines that prevent the OS's page replacement policy from replacing a page of memory in use by a portion of the operating system, such as a device driver. Finally, the module loads the VMM into the allocated memory (414). This step can be performed by the kernel module directly opening the file containing the VMM and issuing the calls to load the VMM into memory. Instead, the kernel module may employ an application outside the OS to load the file. This application could load the file into its own address space in the usual way,

then make a call to the kernel module passing a pointer to its copy of the VMM image. The kernel module could then copy the VMM image into the region of memory it allocated and pinned for the VMM.

[0036] The VMM may handle memory as follows. When an operating system boots on the VMM (see, e.g., FIG. 2b), the OS claims all of the physical memory that the firmware, boot loader, or VMM exposes to it. So that later OS instances have memory to use, the firmware, boot loader, or VMM sets aside memory partitions for later use by other OS instances. Alternately, the VMM gains control over the running OS's use of the memory. Once it has that control, the VMM can take memory away from the first OS instance that booted for use by other OS instances.

[0037] To partition memory for later use by the VMM and operating systems, the boot loader, firmware, or the VMM (if the VMM gets control before the first OS to boot) can modify the table passed to the operating system describing the memory that the OS can use. The table may be modified to expose to the booting OS one partition of memory. The table also may be modified to hide from that OS the memory dedicated to the VMM, and each partition of memory that will be provided to another OS instance.

[0038] If an OS boots on the hardware, and claims all the memory in the hardware (see, e.g., FIG. 2b), the VMM performs steps to gain control over that memory. As a first example of gaining control over the memory, the VMM may perform runtime virtualization of the physical memory used by the OS. Such runtime virtualization is disclosed in assignee's U.S. Serial No. _____ filed _____ (attorney docket no. 200300561-1) and incorporated herein by reference. Because the CPU is already virtualized, the VMM already has control over the hardware to perform the method disclosed therein.

[0039] As a second example of gaining control over the memory, the VMM may use a device driver or kernel module in the OS to "borrow" memory from the running OS for use by other OS instances in a manner similar to the one depicted in FIG. 4. The VMM can invoke a routine in the kernel module requesting the module to allocate memory using the kernel's internal memory allocator. The module pins that memory to prevent the OS from taking it away, and then hands that block of memory to the VMM. The VMM can then expose that memory to a second OS instance during its boot. Alternately, the VMM can reassign the memory from one running OS instance to another. Using a kernel module in the receiving OS as well, the module would add the pages of memory in the block directly to the OS's "free list" of pages, and increment appropriately its count of free pages in the system. If one instance has borrowed memory from another instance, when that one instance is ready to return memory to its place of origin, the kernel module can allocate pages, decrement the free page counters, and hand the pages to the VMM to return.

[0040] The VMM can also perform steps to gain control of the I/O. As a first example, the VMM can virtualize I/O devices at runtime by commencing I/O emulation at runtime as described in U.S. Serial No. _____ filed _____ (attorney docket no. 200309154-1) and incorporated herein by reference. Because the CPU is already virtualized, the VMM already has sufficient control over the hardware to perform the method disclosed therein.

[0041] As a second example of interposing the VMM on I/O, the operating system is provided with "dual-mode" drivers. The dual-mode drivers perform direct hardware control in "native" mode and communicate with device drivers of the VMM in "virtual" mode.

[0042] For example, consider a dual-mode network card driver whose "send" routine is called. If the mode bit is set to "native", that driver would enqueue the message on its queue of outgoing packets, and eventually issue direct I/O instructions to hand the packet off to the network card for sending. If the mode bit is set to "virtual" the driver would instead pack up the message and invoke the corresponding device driver maintained by the VMM. The corresponding VMM device driver would call its own send routine to send the message. The VMM send routine would then enqueue the message and eventually perform the I/O instructions needed to send the message. Each native device driver in the VMM has a routine for importing the state of the corresponding driver maintained by the operating system, and exporting its state to the corresponding OS driver. When interposing a VMM, the state maintained by the OS's device driver (if any) would be handed off to the VMM's driver via one of these routines.

[0043] If the dual-mode network card driver receives a "switch mode to virtual" call, it could delay the processing of new messages while finishing I/Os that have already been enqueued (if draining the queue simplifies the mode switch). Then, if needed, the dual-mode driver could call a routine in the corresponding VMM driver to export to that driver any state the OS driver was maintaining (for some devices or drivers, there may not be any such state). The dual-mode driver could then set its virtual/native mode bit to "virtual", and resume processing messages by forwarding new requests to the appropriate routines in the VMM's native device driver.

[0044] Reference is made to FIG. 5, which illustrates how the dual-mode drivers may be used to interpose the VMM on an I/O device. First, interrupts are disabled (510). Next, the dual-mode driver for this device in the OS is instructed to switch to the virtual mode of operation (512). The driver

can be so instructed by the kernel module in the OS, by the VMM already running on the CPU, or by an application that calls an IOCTL routine in the driver. Next, interrupts for the device are redirected to handlers in the VMM, if they were not already redirected when the CPU was virtualized (514). Finally, interrupts are re-enabled (516). From that point on, the OS's dual-mode driver performs I/O by calling the VMM's driver for the device.

[0045] Devirtualization will now be discussed. The VMM can devirtualize one or more of the CPU, memory, and I/O devices. If the CPU is devirtualized, then both memory and I/O are also devirtualized. If the CPU remains virtualized, the memory alone can be devirtualized, or I/O alone can be devirtualized, or both memory and I/O can be devirtualized.

[0046] Reference is now made to FIG. 6, which illustrates how the VMM can return control of an I/O device to the OS via a dual-mode device driver. The steps in FIG. 5 are reversed. First, interrupts are disabled (610). Next, the dual-mode driver for the I/O device is instructed to switch to the native mode of operation (612). If the dual-mode driver receives a "switch mode to native" call, it could query a routine in the VMM's native driver to import state that driver has been maintaining (if any). That routine would also let the dual-mode driver know that it was safe to start directly issuing I/Os to the I/O device. Next, interrupts for the I/O device are redirected to handlers in the OS (614). Finally, interrupts are re-enabled (616). The dual-mode driver can then begin processing device requests, enqueueing them as needed, and directly issuing I/Os to the I/O device to fulfill those requests, without going through the VMM driver.

[0047] If dual-mode drivers are not used, the VMM can devirtualize the device by ceasing emulation of the I/O device at runtime as disclosed in U.S. Serial No. _____ (attorney docket no. 200309154-1).

[0048] The VMM may devirtualize memory as disclosed in U.S. Serial No. _____ (attorney docket no. 200300561-1). The VMM should return control of devirtualized memory to the OS.

[0049] If the VMM took control of memory using a special kernel module that could borrow blocks of memory, by the time only one OS runs, all memory used by other OS instances should have been returned by the VMM to the one remaining OS. Similarly, if memory was partitioned at boot time for this OS, the OS already controls its memory partition; thus there is no memory for the VMM to return.

[0050] Reference is now made to FIG. 7, which illustrates an exemplary method of devirtualizing the CPU. When devirtualization commences, only one OS is running on the VMM. The steps of FIG. 3 are reversed. Interrupts are disabled (710), a VMM shutdown routine (if any) is invoked (712), interrupts are redirected to handlers in the OS (714), and interrupts are re-enabled (716). If the OS addresses physical memory, the addressability of physical memory is enabled (718).

[0051] Next, privileged instructions are caused not to trap to the VMM (720), and control is returned to the OS (722). On some architectures, causing the privileged instructions not to trap to the VMM may involve restoring the "normal" processor mode (privilege level) of the CPU for the OS. In typical virtual machine systems, the VMM runs in the most privileged processor mode, and it makes the OS run in a less privileged processor mode. By restoring the normal processor mode of the OS (720), the VMM allows the OS to execute without its privileged instructions trapping to handlers in the VMM. In such systems, completing these last two steps (720-722) can involve merely returning control to the OS while leaving the CPU in the processor mode normally reserved for the VMM. In other systems, the

VMM may need to set the CPU to a different processor mode before returning to the OS. In systems where the OS runs at a reduced privilege level even when not on a VMM (such as systems based on the HP Alpha architecture), the VMM may not have to restore the normal processor mode. In certain systems, the VMM may modify the OS's executable image in memory. For example, the VMM may modify some of the OS instructions as an optimization, or to cause the OS instructions to trap to the VMM. In these certain systems, the VMM may restore the normal executable image of the OS during step 322 of devirtualization.

[0052] Reference is now made to FIG. 8. After the hardware has been fully devirtualized, the VMM's activities are completely stopped, control of the hardware is passed to the OS, and the VMM can be removed from the system. First, control of memory is returned to the OS (812). Next, control of I/O devices is returned to the OS (814). Next, the CPU is devirtualized (816). It may be desirable to keep the VMM loaded in memory to enable it to quickly virtualize the hardware again later. However, if the memory used for the system was borrowed from an OS, the VMM may also be unloaded by returning the memory to that OS for its use. This last step will likely be carried out using a kernel module or device driver within the OS, as described above.

[0053] The present invention is not limited to the specific embodiments described and illustrated above. Instead, the present invention is construed according to the claims that follow.